Corbin Mayes - 2/24/19

Hidden-Markov Model Assignment

Maze Generation

I started off this project by creating a maze generator because hand-writing each maze file is tedious and not worth the effort. In the initializer of the function, it takes the given width and height of desired maze otherwise it creates a 1 by 1 maze (not very exciting but maybe someone will want to explore this).

generate_maze function

> This iterates through two for-loops, one representing the width and the other height for the maze. The function picks a random symbol from the color list to add the maze string. To create a maze with no walls I just removed the wall symbol from the list.

maze_file function

> This opens a file under the given file name or generate a new one. It then writes the generated maze string to the file and closes it.

HMM

To begin this program, I did not know what I would need for global variables so I simply started off with a maze global variable and began by reading in the maze file and transforming it in to something that is easier to use.

read_maze function

> This function begins by opening the file that is passed into it. It then reads the lines and closes the file. For each spot in the maze, it adds the one to the maze list that numbers from the top left corner to the bottom right and adds the corresponding color for that spot into a dictionary for later use. This also calculates the width in the iterations and then finds the height of the maze as well.

After reading the maze, I wrote the functions that generate the initial model (which I originally thought was meant to be the transition model) and the sensor model

generate_initial_model function

> If the maze is not empty, it first starts off by counting all the available spaces (spaces that are not a wall). I realize that I could have done this more efficiently creating a global variable that when I read in the maze it also counts the number of walls or the overall number of available spaces but because I don't use this number anywhere else I just created a separate for-loop for it. It then iterates over every space and adds the space to the dictionary along with the probability that the robot starts in that space which should be equal for every open space.

generate_sensor_model function

> This function generates a dictionary that has the colors of the maze as the keys and list of probabilities as the values. The list is a list that contains the probability that given a sensor color (the key) that the robot is on that given space. It is 88% if the sensor matches the space's color otherwise it is 4%. If the space is a wall then the probability is 0%.

Because I still had the misconception of what the transition model was meant to be I thought this was all I needed before I began the forward filtering initializer and recursive functions. I also created the get_prob_distribution function right after so that the result was easier to read and understand for later decoding.

forward_filtering function

> This is the initializer function. It calls and returns the timestep function which is the recursive part of forward filtering algorithm. It passes the initial model into it for the first step to use.

timestep function

This function takes a transition model, the current time step, and the probability distribution up to the current call. If the time steps are greater than the length of the sensor reading list then it returns the probability distribution and ends the recursion.

Predict Step

This begins the process of creating the probability distribution for the current step. It multiplies the current transition model by the global transition model. This results in a list of lists. Then it adds the columns of the list and produces a dictionary with spaces as keys and the results as the values.

Update Step

This is the second step of the recursive call. It takes the dictionary from the predict step and multiplies each value by the sensor model and adds it to another dictionary. It then calculates alpha which is found by adding up all the probabilities within the dictionary and dividing one by the sum. Then to normalize the dictionary values, it multiplies each one by alpha.

After the probability distribution for the timestep is calculated using the above steps, it is added to the dictionary with the time as the key and the probability distribution as the value. Then the recursive call is made with the new probability distribution becoming the transition model for the next step, the time plus one, and the updated dictionary with the new probability distribution.

get_prob_distribution function

This just transforms the hard to read dictionary into a string that is more easily presented. It separates each time step and labels the positions as well as changes the probabilities into percentages so that debugging and reading the information is easier to understand

After writing all these functions, I realised that the filtering functions weren't updating the probabilities properly. This is because of the misconception that the initial model is the transition model. So I wrote the generate_transition_model function as well as its helper function get_spaces_next_to. Finally I added the function generate_sensed which creates a random sensor reading for the hmm to use so a new one does not have to be inputted every time.

generate_transition_model function

This function starts by getting all the surrounding spaces for each space in the maze. To do this it calls the get_spaces_next_to helper function

get_spaces_next_to function

Because of the way I classify each spot in the maze by a single number instead of an x,y pair, it adds some difficulty in finding the spots that around a given space. To get the spaces in horizontal proximity is easy because you can add or subtract one from the current space. To test if this is within bounds, by knowing the width of the maze it's possible to test if the space by dividing the space by the width and getting the remainder. Because we know the width we can also get the vertical proximity spaces by adding and subtracting the width of the maze. To test if these are in bounds, it sees if the new space is either less than 0 or greater than the maze length. If a space meets all these conditions as well as not being a wall it is added to the list which is returned at the end of the function

After calling the get_spaces_next_to function, depending on how many spaces are returned for the given space results in the probability that the robot remains in the same space. It then adds 0.25 chance for each space in the list and 0 for the rest of the spaces. This is the transition model that represents the spaces a robot can move to given the space it is in.

HMM_Run

This is the run program for the HMM assignment. It contains the code to run all the current maze files and generate a random sensor path to follow for each maze. It makes seeing the test code easier and keeps it all in one place thats easy to find.